# Continuation based C の GCC 4.6 上の実装について

大 城 信 康† 河 野 真 治†

GCC-4.6 をベースとした CbC コンパイラの実装を行った. CbC のコンパイラは GCC-4.2 ベースのコンパイラが 2008 年に開発されており, 以来 GCC のアップデートにあわせて CbC のコンパイラもアップデートが行われてきた. 今回は GCC-4.6 への実装を行った. 本論文では GCC-4.6 への CbC の具体的な実装について述べる。

# The implementation of Continuation based C Compiler on GCC 4.6

Nobuyasu Oshiro† and Shinji Kono†

We implemented Continuation based C Compiler on GCC-4.6. CbC Compiler on GCC-4.2 was developed on 2008. Since then we kept to update it. In this paper, we introduce implemented Continuation based C Compiler on GCC-4.6.

## 1. 歴史的経緯

当研究室では, 継続により処理を行うプログラミング言語 Continuation based C (以下 CbC) を開発している. CbC の構文は C と同じであるが, 継続によりループ制御や関数コールを取り除かれる.

2008 年の研究において GCC-4.2 ベースの CbC コンパイラが開発された. 以来,GCC のアップデートに合わせて GCC ベースの CbC コンパイラのアップデートを行って来ている. お陰で,GCC の最適化やデバッガの機能を使うことができより実用的な CbC プログラミングが行えるようになった.

しかし, 未だに GCC ベースのコンパイラには幾つかのバグがある. 今回,GCC-4.6 への実装も兼ねながら問題の部分の改善を行った. 本論文では, CbC,GCC の簡単な説明と,GCC-4.6 への実装を具体的に述べる.

## 2. Continuation based C (CbC)

Continuation based C (以下 CbC) は当研究室で開発しているプログラミング言語である. 構文は C と同じであるが, ループ制御や関数コールを取り除き継続 (goto) を用いている. また, コードセグメント単位で処理を記述するという特徴がある. 図 1 は CbC におけるプログラムの処理の流れを表している.

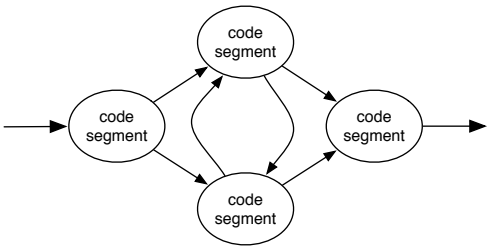


**図 1** コードセグメント間の継続 (goto)

### 2.1 継続 (goto)

コードセグメントへと移った処理は C の関数と違って呼び出し元の関数に戻ることはない. コードセグメントは自身の処理が終われば goto により次のコードセグメントでの処理に移る. goto によるコードセグメント間の移動を継続と言う.

### 2.2 コードセグメント (code segment)

CbC におけるプログラムの基本単位としてコードセグメントという概念がある. コードセグメントの記述の仕方は C の関数と同じだが, 型に“ __code ”を使って宣言を行うところだけが違う. 関数と同じように引数を持たせて継続させることもできる. しかし, 関数とは違ってリターンを行わない為返り値を取得することはできない. 図 2 は CbC で書いたプログラムの例である. 与えられた数 x の階上を計算して出力するプログラムとなっている.

† 琉球大学
University of the Ryukyu

```
__code print_factorial(int prod)
{
  printf("factorial = %d\n",prod);
  exit(0);
}
__code factorial0(int prod, int x)
{
  if ( x >= 1) {
    goto factorial0(prod*x, x-1);
  }else{
    goto print_factorial(prod);
  }
}

__code factorial(int x)
{
  goto factorial0(1, x);
}
```

**図 2** CbC のプログラム例

## 3. Gnu Compiler Collection

GCC-4.6 への実装の前に,GCC によるコンパイルの一連の流れについて触れておく.

### 3.1 3 つの中間言語

GCC は内部で Generic Tree, GIMPLE, RTL の 3 つの中間言語を扱われる.

#### 3.1.1 Generic Tree

まず,GCC で読み込まれたソースコードは Generic Tree 呼ばれる構文木のデータ構造で表される. 図... に Generic Tree で表現された例を示す.

#### 3.1.2 GIMPLE

Generic Tree により表現されたデータは次に GIMPLE という構文木へと変換される. GIMPLE は Generic Tree より制約がかかった状態で作成される. 制約は「1 つの枝に 4 つ以上の子を持たせない」といったもので, GIMPLE へと変換されたデータは Generic Tree より簡単な命令で表されることになる.

#### 3.1.3 RTL

Gneric Tree から GIMPLE, そして RTL へとデータは変換され最後にアセンブリ言語で出力される.

## 4. GCC-4.6 への実装

### 4.1 Tail Call Elimination

CbC の継続の実装には GCC の最適化の 1 つである Tail Call Elimination (末尾除去) が使われる. Tail Call Elimination とは関数の最後の処理で別の関数呼び出しを行った際に, call ではなく jmp を用いて大元の関数へ戻るようにする最適化のことである. 図**??** は Tail Call Elimination が行われた際のプログラムの処理を表している.

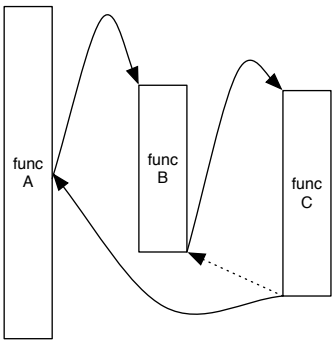


**図 3** Tail Call Elimination

#### 4.1.1 expand_call

### 4.2 引 数 渡 し

通常コードセグメントの継続において, 引数は C の関数と同じスタックを用いて渡される. GCC には引数渡しをスタックではなくレジスタを用いて行う機能として fastcall がある. fastcall を用いてコードセグメントを宣言することで, レジスタを用いた速度の向上を図る.

#### 4.2.1 fastcall

コードセグメントの引数渡しを fastcall によりできるだけレジスタを用いて行うようにする. C において fastcall を用いる場合は関数にキーワード“ __attribute__ ((fastcall)) ”をつけて行う. だが, コードセグメントを全てこのキーワードをつけて宣言することは実用できではない. そこで, コードセグメントで宣言された場合,fastcall が自動で付くように実装を行う. 図 4 はコードセグメントに fastcall 属性を付与しているソースである.

```
if(!TARGET_64BIT) {
  attrs = build_tree_list (get_identifier("fastcall"), NULL_TREE);
  declspecs_add_attrs(specs, attrs);
}
```

**図 4** fastcall 属性付与

if 文で条件を決めているのは,64 bit の場合 fastcall が標準で行われ為である.

## 参 考 文 献


1) 河野真治: “ 継続を基本とした言語 CbC の gcc 上の実装 ”. 日本ソフトウェア科学会第 19 回大会論文集, Sep, 2002
2) 河野真治: “ 継続を持つ C の回言語によるシス

テム記述 ”. 日本ソフトウェア科学会第 17 回大会論文集, Sep, 2000
3) 与儀健人, 河野真治: “ Continuation based C コンパイラの GCC-4.2 による実装 ”. 琉球大学　情報工学科 学位論文, 2008
4) 与儀健人, 河野真治: “ 組み込み向け言語 Continuation based C の GCC 上の実装 ”. 琉球大学大学院　理工学研究科 学位論文 (修士), 2010
5) 下地篤樹, 河野真治:“ 線形時相論理を用いた Continuation based C プログラムの検証 ”. 琉球大学大学院　理工学研究科 情報工学専攻 学位論文 (修士), 2008
6) 楊挺, 河野真治: “ Continuation based C の実装 ”. 琉球大学大学院　理工学研究科 情報工学専攻 学位論文 (修士), 2002
7) GNU Compiler Collection (GCC) Internals: “ http://gcc.gnu.org/onlinedocs/gccint/ ”